1985

Some Thoughts on a Uniform Generic Command Interface

Balachander Krishnamurthy

Report Number:
85-513

Some Thoughts on a Uniform Generic Command Interface

Balachander Krishnamurthy

April 1985

CSD-TR-513

Department of Computer Science
Purdue University
West Lafayette, IN 47907

* This work was supported in part by grants from the National Science Foundation (MCS-8219178), SUN Microsystems Incorporated, and Digital Equipment Corporation.

Some Thoughts on a Uniform

Generic Command Interface

ABSTRACT

We study the usefulness of having a uniform command interface with the help of generic commands. Generic commands are a small set of commands that are available globally and cause the right action to be performed in the right place. These commands result in the user having to remember only a basic set of natural commands. The choice of key bindings to the generic commands is under the control of the user while the generic commands themselves are bound to interface functions by the interface programmer. Greater flexibility with more customization is thus obtained. We consider generic commands as a generalized *Do What I Mean* [DWIM] mechanism. Local abbreviations and global flags, over which the user has control, are also considered in the context of generic commands. The concept of generic commands is examined on a system-wide basis.

Keywords and Phrases:

Generic commands, Key-bindings, Command Interfaces, Modes, Workstation, Extension Language, DWIM

1. Introduction

Much effort has been made to improve the interface provided to users of various systems. Of late a concerted effort has been made to give more attention to the user interface when a new system is designed [DeT80, Lam83]. A common pitfall of a programming environment is its inherent complexity combined with the lack of a uniform interface. Furthermore, addition of *ad hoc* features to improve the interface only increases its complexity. We explore an alternative way: unify the existing interfaces into a more uniform one.

We approach the problem of a uniform interface with the idea of generic commands as an integral part of a recognizable front end to the computing environment. This research is part of the DASH (Dynamic Access to Shared Hosts) [Kor84] project, which is the intelligent terminal portion of the TILDE [CKT84] research project. DASH uses the workstation as an agent, to access network services and to interact on behalf of the user with programs running on one or more remote host processors. We first give a brief introduction to the notion of generic commands by considering the work that has already been done in this regard.

In their paper "Structured Graphics for Distributed Systems" [LaN84], Lantz and Nowicki lament the absence of state-of-the-art command interfaces. They point out the need for *higher level short circuiting* via generic editing facilities. The concept of generic commands is not entirely new [Car82, SIK82b, Weg84]. According to Wegmann [Weg84], generic commands are a small set of commands that can be used throughout the system. Smith et al. [SIK82b] give a more precise definition when they describe generic commands as a means to get at the underlying principles by stripping away the extraneous application-specific semantics. A related paper [SIK82a] clearly outlined the need for generic commands. However, stress was laid on one particular command (*move*) as the context was an object oriented architecture. We consider generic commands on a broader scale without narrowing our view to a particular application area. We define generic commands as the small set of commands that can be used anywhere in the programming environment with a fair expectation of the "right" thing to happen in the right place. A basic premise

is that a new user will try the command expecting that command to perform some action and it actually does. This has been more colorfully described as the *law of least astonishment* [DeT80]. We also hope to free the user from having to remember several dozen commands, the functionality of which are similar and can be abstracted.

Our view of generic commands has three conceptual layers:

- input event layer
- generic commands layer
- interface function layer

The three–level approach is used to allow the user to rebind the generic commands to any desired keystroke and at the same time to allow the user to write new functions to which he can bind keys. We differentiate between users and programmers as follows: users are more likely to change the bindings of the keystrokes while programmers are more likely to change existing functions or write new ones. Of course, the ability to write new functions brings the added responsibility of having to decide to which generic command it should be bound.

The example programming environment being considered here is EMACS [Sta81]. EMACS (Editing Macros) is a widely-used extensible, customizable editor. EMACS is an evolving programming environment. It has an extension language MLisp (Mock–lisp, a lisp–like language) that allows access to basic editing operations and a set of programming primitives for combining them into functions that can be bound to keystrokes. These functions are executed whenever the corresponding keystrokes are typed. We add a new layer (that of generic commands) between the keystrokes and the functions.

In the rest of the paper we look at the motivation for generic commands and a local implementation before going on to see how this concept can be generalized beyond the editor.

2. Motivation

Dijkstra [Dij76] characterized the structure of programming language con-

structs in terms of loops, assignment statements, and recursive calls. Winograd [Win79] added processes to this model and stressed the need for "combining objects into a *structured object.*" We consider generic commands to be (abstract) structured objects whose components are functions that are selected on the basis of the mode. Our idea is to make use of a few simple, easily remembered commands to specify functions across a wide variety of modes, where a mode can be considered to be an environment or a context. While the precise function of the generic command might vary with the current context it would still be what the user might consider natural [Lam83].

We will pick an elementary set of generic commands and try to model most of the applications around these. Because the model is incremental we can add new generic commands, bindings to these commands, and possibly new functions.

One example of a generic command is *next*. Depending on the context that the user is currently in, *next* can be bound to the appropriate function. Suppose the user is perusing his mail messages. The generic command *next* could be bound to a function that not only moves on in the list of messages but displays the next message as well. When editing, the same command *next* moves the cursor to the next line.

The keystrokes need only be bound to the generic commands and not to any explicit functions that actually carry out operations. Thus, changing context does not necessarily mandate rebinding. We have thus separated the syntax and semantics of the binding issue to free the user from having to worry about the semantics. A distinction between novice and advanced users has to be made here: novice users normally do not write new functions; this is usually the task of the programmer or the advanced user. The programmer can take advantage of the functionality of the generic command in the current context to alter the function or create one if it does not exist.

The central idea of the DASH generic command interface is a generalization of the DWIM [Tei84] idea. Teitelman's stress on DWIM was more on automatic error correction and implicit assistance. No attempt was made to simplify the binding

mechanism between the keystrokes and the functions. We bind generic commands to function names in a uniform manner. Users have to remember only a small set of commands but can make use of a large set of operations.

Extreme examples of the lack of uniformity and clarity in existing systems abound. As an example, in EMACS the key-binding ^X-^P is bound by default to the function that copies a range of characters to a file (*write-region-to-file*) in most modes except in mail-handling mode (*mh-mode*) where it updates the list of mail messages received (*pack-folder*). The two functions are radically different having no functionality whatsoever in common. If we had a generic command *copy* we would expect it to copy the region to a file when we are editing a file, copy files in *dired-mode* (directory editing mode) and copy a message to another folder in *mh-mode*. Similarly, the generic command *update* would update the folder by removing the marked headers in *mh-mode*, write out a modified text file to backup storage when we are editing a file. In other words, the user is aware that he is indeed updating some information, whether it is a file that has been modified, a set of messages that has been scanned and marked, or a source program that has been modified and needs to be compiled.

The problem is not the obscure bindings of keystrokes to functions, but the fact that the way to specify a particular operation is shrouded in a mysterious command syntax. To remove the shroud we should resort to better and more meaningful commands or keystrokes. The DWIM mechanism, although powerful in its own right, is not general enough to handle this problem. The DWIM mechanism, which is implemented as a single key (the DWIM key), usually performs the most appropriate thing in the current context. The limitations of the DWIM mechanism lie in the fact that it is capable of doing *only one* "right" thing. The DWIM key should do the "most appropriate" thing at any given instant. The most appropriate thing clearly varies with the user. If we are leafing through a mail folder we can use the DWIM key to read the next message. Should the current message be deleted or should it be left alone? Should we return the user to whatever he was previously doing if we are at the end of the message folder? If we attempt to incorporate all

of this into the DWIM key it would soon be overloaded with details and lose its generality. Already the DWIM key maintains context and some history in order to decide what is the right thing to do. If we force it to look at the mode, the local keybindings and the local flags, we will not only have an overloaded DWIM key, but a difficult one to implement as well. Rather than vary the meaning of the DWIM key with the idiosyncrasies of the user, it is more appropriate to provide the user with a uniform set of generic commands that he can bind to keys of his choice and do all the "appropriate" things just as easily.

The drawback of DWIM lies in the fact that it is limited to one key – generic commands provide many DWIM keys without forcing the user to pay a significant price in having to remember strange commands. The crucial point to observe is that the functionality of the generic command concept is different from that of DWIM. DWIM is meant to be more of an automatic error corrector while generic commands are not involved in the syntactic part of the interaction with the user. Generic commands are similar to DWIM in the sense that they aid the user with the facility to select the appropriate function in the current context. However, it goes beyond providing a simple *Do What I Mean* facility by making the correct function available to the user in the present context. As there are quite a few functions that are available to the user in any given context it is hard to satisfy his needs with a single DWIM key. The feasibility of the generalization of DWIM can be seen here – the DWIM mechanism can be overlaid with the generic command mechanism.

We will now examine what the user has to know about generic commands, what control he has over them, and where he can use them. The user can rebind keys to any or all of the generic commands. In turn, he can rebind the generic commands themselves to different functions by incorporating his own library routines or by changing the existing ones. If a particular generic command, for example, *next*, is not to his liking and he does not wish to use it, he can use the command by specifying the full name (e.g., *mh-next-line*). However, the user is more likely to use *next*. *next*, being a generic command, is not restricted to mail-handling mode alone. It can be used, for example, when an entire directory is being edited (*dired*),

or when editing ordinary text.

3. Beyond the editor

The universality of generic commands may not be clear due to the constant stress that has been made so far on the editor. We can easily extend the idea beyond the editor. A good example is David Korn's UNIX command interpreter, called *ksh* [Ste84]. *Ksh* has incorporated the features of the Bourne Shell [Bou78], the original command interpreter of the UNIX operating system [RiK74] and some of the features of the C-shell [Joy83]. The Korn shell maintains a history of the commands typed and permits command line editing. The user can specify which editor his command line manipulations should be patterned after. All the commands that are used in the editor specified can then be used to manipulate the command line. This unification of the editing environment and the command line environment simplifies modification of previous commands that the user has typed. This is also useful if he wants to execute a command repeatedly.

For example, to redisplay the previous command that was typed to the shell the user types the equivalent of the *previous line* command of the editor. He could then manipulate the command line using the editing commands of the specified editor. In short, it provides him with a one line window where he can use his editor commands. In our view this is a generic command facility on a limited basis. The user's ability to redisplay the previous line is making use of the generic command *prev* – which in command mode redisplays the previous command in his command history list. His manipulations of the command line are possible since the editor commands are available on a system wide basis. Modifying the shell to unify it with the other parts of the programming environment is a growing trend shown by another recent system WSH [BBM84]. In this system, windows represent virtual terminals and the shell is combined with a window manager. This does not, however, attempt to provide a uniform interface globally.

4. Other applications of generic commands

In this section we show how the introduction of generic commands can help make the interface to other system features clearer. In most programming environments there are user configurable parameters called flags. Flags can be viewed either as a toggle mechanism or a variable which usually has one of a few values. Here we consider the variables (flags) that are defined within EMACS which can be set at startup time and can also be altered dynamically. They are essentially configuration parameters. Flags are used to specify several things: a typical example is *scroll-step*, which is used to specify the number of lines the window must be scrolled when the point of insertion moves out of the window. This variable can be set to the appropriate value depending on the mode. By default it may move the insertion point to the middle of the window but if the current major mode is *command-mode*, it should probably move one line down. In other words, these flags should not be global: they should have buffer and mode-specific values. The generic commands can aid in this. For example, *next* may cause the value of the *scroll-step* to be reset to the appropriate value depending on the mode in which *next* gets executed.

EMACS is a fairly large package and as each flag has varying degrees of usefulness most users need not be (and are not) aware of them. While there are a few local flags (i.e., mode-specific), the flags most commonly used suffer from an affliction, viz., they are global. While it is true that some flags do deserve to be global (e.g., *silently-exit-emacs*) there are several flags which should not be global – in fact they should change with the mode and be the "right" one for that particular mode. For example, *wrap-long-lines* is used to specify whether a user wants long lines to be clipped or not. If the lines are not clipped they are displayed in a wrapped around fashion with a marker at the end of the long line. If a user is in handling mail, he would usually like the flag *wrap-long-lines* to be turned on to prevent words from being clipped off the end of the screen. At the same time the buffer which displays the header of the mail (usually consisting of sender, date and probably first line of message) should probably have its long lines clipped, not wrapped. Of course, we could modify the MLisp library routines to effect this change. This would imply

that a lot of flag changes (possibly needless) would be made every time we change mode. The real problem is that that these flags are global.

We demonstrate the usefulness of the generic command *create*. Depending on the current context, *create* creates a buffer with certain attributes set. For example, if *create* is invoked when the major mode is *text-mode*, the new buffer will automatically inherit the characteristics of this mode. Thus, all the abbreviations that are valid in *text-mode* are turned on, the space character gets redefined so that one can continue to type without pressing return and the lines will not extend past a predefined margin. The benefits of this command can be seen in other contexts as well. Suppose the user is processing mail messages. A fairly common action is *replying* and instead of having an extra command called *reply* we suggest that the user use *create*. In the present context *create* will automatically set up the header for the reply message (including things like who the reply is for and a brief reference to the message being replied to). If the user is looking at the list of mail messages (as opposed to a particular message), invoking *create* will cause an empty draft to be created. This brings out another facet of generic commands, the elimination of needless mode-specific commands. Commands like *reply* (used to reply to a specific message) or *compose* (used to compose a message to be sent to someone) can be eliminated if the globally available generic commands are used. When we advocate generic commands, we are not introducing a new set of commands. On the contrary, we eliminate a plethora of commands and replace them with a succinct set of commands that are widely usable.

If the present context is *command-mode* then *create* attempts to edit a file (whose name is required as an argument). This brings us to the other important point that has not been mentioned so far: generic commands, like any other commands, take arguments at times. These arguments are also dependent on the context. Thus, if *create* is invoked in command mode the user is prompted for an argument – viz., a file name to be edited or created.

Another example is the generic command *help*. Novice users use this to get information about the available commands and expert users use this to find out the

parameters of a certain procedure. The generic command *help* is mapped to the system command for help (i.e., *man* in UNIX) in command mode. In other contexts it gives a brief summary of valid commands. As a more precise example, in EMACS *help* is bound to the *describe-word-in-buffer* command which in specific language contexts returns the type of the function and its arguments.

Some of the generic commands we are considering and the places they are most likely to be used are shown in the appendix.

5. Future Directions

Another aspect we intend to explore is the interaction of the style modules [Car82] with the extension language and the generic command mechanism. Carey has defined interaction style as the user's perception of a dialogue with the computer which includes interaction techniques and display layout. Interaction style can be considered at both the syntactic and semantic levels, and specifications can be made in corresponding modules. We believe that style modules have a close relationship to the generic command mechanism. The user can specify his choice of interfaces not only to the editor but to the entire system. Carey considers style mainly at the semantic level where, for example, the users can specify if they would prefer process oriented or result oriented requests. The example cited there is that of an editor always being in input mode (like EMACS, which is why it is considered to be a modeless editor). This style is result oriented, because we see the result of our keystroke instantly.

Further extensions will be in places where the abstract concept of generic commands can be used. We consider the example of keybindings and functions to be just one application. Other applications can be in the area of database queries (a generic query will be translated into an appropriate query depending on the relation).

6. Conclusion

The conclusions that were reached at the end of our study clearly showed that

there were indeed severe inconsistencies in the way command syntax had evolved and the way bindings were done. The user should have control over his programming environment; he should be able to change the default bindings to suit his liking, and should not be required to remember strange and obscure keystroke bindings. The three–level approach of binding keystrokes to generic commands and binding generic commands to functions provides a uniform and simple interface to the system.

Acknowledgements

This idea is primarily due to my advisor John T. Korb and is part of the DASH project conceived by him. I would like to express my gratitude to Kathleen Korb and Douglas Comer who patiently read preliminary versions of the manuscript and offered several useful suggestions. Also instrumental in this work are the students in the DASH seminar, notably, Thomas Narten, Francie Newbery and Craig Wills.

References

[Bou78] S. Bourne, UNIX Time-Sharing System: The UNIX Shell, *The Bell System Technical Journal* **57**, 6 (July-August 1978) 1971–1990.

[BBM84] J. B. Bresnahan, D. T. Barnard, and I. A. Macleod, WSH — A New Command Interpreter for UNIX, *Software–Practice and Experience* **14**, 12 (December 1984) 1197–1205.

[Car82] T. T. Carey, A workstation for interaction styles, *IEEE COMPSAC Proceedings*, 1982, 303–307.

[CKT84] D. Comer, J. T. Korb, W. Tichy, and T. Murtagh, The TILDE project, Computer Science Dept. Tech. Rep. 500, Purdue University, November 23, 1984.

[DeT80] L. P. Deutsch and E. A. Taft, Requirements for an Experimental Programming Environment , CSL-80-10, Xerox Palo Alto Research Center, June 1980.

[Dij76] E. Dijkstra, *A discipline of programming*, Prentice–Hall, 1976.

[Joy83] W. N. Joy, Introduction to the C-Shell, in *4.2 BSD Reference Manual*, University of California, Berkeley, August 1983.

[Kor84] J. T. Korb, An Overview of the DASH Intelligent Terminal Project, Computer Science Dept. Tech. Rep. 492, Purdue University, Department of Computer Science, September 1984.

[Lam83] B. W. Lampson, Hints for Computer System Design, *Proceedings of the Ninth Symp. on Operating System Prin.*, October 1983, 33–48.

[LaN84] K. A. Lantz and W. I. Nowicki, Structured graphics for distributed systems, *ACM Transactions on Graphics*, January 84, 23–51.

[RiK74] D. Ritchie and K.Thompson, The UNIX Time-Sharing System, *Communications of the ACM* **17**, 7 (July 1974) 365–375.

[SIK82a] D. C. Smith, C. Irby, R. Kimball, and E. Harslem, The Star User Interface: An Overview, *Proceedings of the AFIPS 1982 NCC* **51**, (1982) 517–528.

[SIK82b] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem, Designing the Star User Interface, *Byte* **7**, 4 (April 1982) 242–282.

[Sta81] R. M. Stallman, EMACS: The extensible, customizable self-documenting display editor, *SIGPLAN Notices Notices* **16**, 6 (June 1981) 147–156.

[Ste84] B. Stewart, Is ksh available/A description of ksh, *Unix-wizards-request@BRL-TGR, Message-Id:254@ho95b.uucp*, November 1984, .

[Tei84] W. Teitelman, A Tour Through Cedar, *IEEE Software*, April 1984, 44–73.

[Weg84] A. Wegmann, Vitrail: A Window Manager for an Office Automation System , *ACM SIGOA Bulletin* **5**, 1–2 (June 1984) 1–12.

[Win79] T. Winograd, Beyond Programming Languages, *Communications of the ACM* **22**, 7 (July 1979) 391–401.

Generic Commands Summary

- *delete:* Delete selected object
- *create:* Create a new object
- *copy:* Copy selected object
- *move:* Move selected object to destination

command	*text-mode*	*mail-mode*	*command-mode*
delete	delete characters, words, lines depending on argument	semi-active command - marks header	removes files, directories, etc.
create	create a buffer to read text in	compose a message (*comp*) or reply	edit a file or create a directory
copy	paste following a delete	refile message to a folder (*refile*)	regular copy of files, directories
move	moves selected characters to insertion point	moves message to folder	moves file to destination directory

- *next:* Move on to next object
- *prev:* Move backwards to previous object

command	*text-mode*	*mail-mode*	*command-mode*
next	next followed by c, w, l, s, p in text mode moves to next character, word, line, sentence, or page	next unmarked message is displayed	argument dependent
prev	analogous to next	previous unmarked message is displayed	displays previous command

- *scroll:* Scroll the objects
- *select:* Select object from a menu or a buffer
- *zoom:* More of the same (previous command)

command	*text-mode*	*mail-mode*	*command-mode*
scroll	move around buffer quickly (beginning, end, arbitrary point in file)	scrolls the mail-messages list	scrolls through history list (preferably in a separate window)
select	select any portion of a buffer (character, word, line etc.)	takes a numeric argument and displays corresponding message	select from a menu
zoom	enlarge or diminish the font size, currently deletes other windows	list all the mail folders in mail-mode, show complete message in *show* mode	magnify size of objects seen on the screen

- *search:* Search for object
- *repeat:* Repeat previous command
- *undo:* Undo previous operation
- *update:* Update current buffer

command	*text-mode*	*mail-mode*	*command-mode*
help	describes the word in buffer using database search	prints out a brief summary of valid commands	prints out a man page of information on command
search	search up/down in current buffer (uses argument facility)	search for string in file containing the messages	search for string or regular expression in file(s)
repeat	repeat search operation in current direction	repeat previous operation (e.g. refiling)	repeat previous command in history list
undo	reverses effect of previous edit operation	marked messages are unmarked	limited use (e.g. *rm* → *unrm*)
update	updates current buffer (writes out file to disk)	scans headers and removes marked messages	"source" files like .login, .cshrc if they have changed